**KOTLIN FUNDAMENTAL**

Kita sudah tahu cara membangun dan menjalankan program Kotlin. Kini di sub-modul ini kita akan mempelajari konsep-konsep dasar (fundamental) pada Kotlin. Kita akan belajar bersama mengenai data types, function, expression dan juga nullability pada Kotlin.

# Hello Kotlin!

Sebelum ke topik-topik fundamental tersebut, mari kita awali dengan pembahasan tentang program Hello Kotlin!. Pada sub-modul sebelumnya kita telah berjumpa dengan sebuah proyek yang menampilkan sebuah teks Hello World! dan juga Hello Kotlin!.

Buat Anda yang pernah belajar pemrograman sebelumnya, tentu tak asing dengan program ini. Hello World! sering digunakan untuk menunjukkan sintaks dasar pada sebuah bahasa pemrograman. Karena kita sedang belajar bahasa pemrograman Kotlin, maka kita mengganti namanya dengan Hello Kotlin!.

Hello Kotlin! merupakan sebuah program sederhana yang digunakan untuk mencetak sebuah teks “Hello Kotlin!” ke dalam layar atau konsol. Berikut adalah contoh kode dari program tersebut:

* // main function
* fun main() {
* println("Hello Kotlin!")
* }

Baris pertama dari kode di atas adalah komentar yang ditandai dengan tanda //.

* // main function

Sebuah komentar akan dilewatkan ketika proses kompilasi, sehingga tidak akan mempengaruhi alur program yang kita tulis. Komentar bisa kita gunakan untuk mendokumentasikan kode yang kita tulis agar ketika suatu saat kita membukanya kembali, kita bisa mengetahui fungsi dari kode yang kita beri komentar tersebut.

Terdapat dua jenis komentar yang bisa kita gunakan. Pertama adalah *single line comment* yaitu komentar satu baris yang diawali dengan tanda // dan berakhir di akhir baris komentar tersebut.

* // single line comment

Yang kedua adalah *multi-line comment* yang diawali dengan tanda /\* dan diakhiri dengan tanda \*/.

* /\*
* multi line comment
* Hello Kotlin
* \*/

Dengan *multi-line comment* kita bisa menuliskan beberapa baris komentar.

Selanjutnya adalah fungsi yang bernama main(), fungsi yang wajib kita definisikan ketika membuat sebuah program. Fungsi main() merupakan sebuah *entry point* yang otomatis akan dipanggil ketika program dijalankan. Pada sub-modul berikutnya kita akan belajar lebih dalam tentang bagaimana mendefinisikan sebuah fungsi.

Kemudian fungsi println(), fungsi yang akan kita gunakan untuk mencetak teks ke dalam layar atau konsol. Fungsi println() membutuhkan satu argumen berupa message dengan tipe data yang dikehendaki. Tipe data yang didukung untuk kita masukkan ke dalam fungsi println() ada di [tautan ini](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/print.html).

Selain fungsi println(), terdapat juga fungsi print() yang berfungsi sama seperti fungsi println(). Bedanya, println() akan menambahkan baris baru setelah selesai mencetak argumen yang diberikan, sementara fungsi print() tidak melakukan apapun ketika argumen yang diberikan, selesai dicetak. Untuk memahaminya lebih dalam, coba jalankan kode berikut:

* fun main() {
* val name = "Alfian"
* print("Hello my name is ")
* println(name)
* print(if (true) "Always true" else "Always false")
* }
* /\*
* output:
* Hello my name is Alfian
* Always true
* \*/

Fungsi println() dan print() secara internal memanggil fungsi system.out.print(message).

* @kotlin.internal.InlineOnly
* public actual inline fun print(message: Any?) {
* System.out.print(message)
* }

Kegunaan utama dari fungsi system.out.print(message) adalah untuk menampilkan pesan yang diberikan ke [*standard output stream*](https://en.wikipedia.org/wiki/Standard_streams). Selain menampilkan pesan yang diberikan secara eksplisit, fungsi tersebut juga dapat digunakan untuk menampilkan nilai dari sebuah *expression* atau variabel seperti yang dicontohkan di atas.

# **Data Types & Variable**

Data types atau tipe data adalah sebuah pengklasifikasian data berdasarkan jenis data tersebut. Untuk mengembangkan sebuah program, ada beberapa tipe data yang akan kita pelajari. Di antaranya adalah **Character**, **String**, **Array**, **Numbers** dan **Booleans**. Semuanya akan kita bahas sejelas dan sesederhana mungkin di dalam sub-modul ini.

Namun sebelumnya, ada satu hal yang kita perlu tahu terlebih dahulu, yaitu **Variabel**. Umumnya variabel digunakan untuk menyimpan informasi atau nilai yang akan dikelola di dalam sebuah program. Sebuah variabel akan membutuhkan kata kunci var atau val, **identifier**, **type** dan **initialization**. Kira-kira strukturnya seperti berikut:

* var identifier: Type = initialization

Berikut adalah contoh variabel dengan tipe String:

* var company: String = "Dicoding"

Mari kita ulas setiap bagian pada struktur variabel di atas.

* **var** **atau** **val**var atau val digunakan untuk mengontrol nilai dari sebuah variabel. Dengan kata kunci var kita bisa mengubah nilai yang sudah kita inisialisasikan. Sebagai contoh:  
  + var company: String = "Dicoding"
  + company = "Dicoding Academy"
* Variabel company yang awalnya memiliki nilai **“Dicoding”** sekarang sudah diubah menjadi **“Dicoding Academy”**. Sedangkan jika kita menggunakan kata kunci val, kita tidak bisa mengubah nilai yang sebelumnya sudah kita inisialisasi. Jika kita memaksa untuk mengubahnya, maka akan terjadi eror seperti berikut:  
  + val company: String = "Dicoding"
  + company = "Dicoding Academy" //Val cannot be reassigned
* **Identifier***Identifier* merupakan nama dari sebuah variabel. Pada contoh kode di atas yang merupakan *identifier* adalah company. Perlu diketahui bahwa di dalam sebuah program kita tidak bisa membuat lebih dari 1 (satu) variabel dengan nama sama.
* **Type**Pada bagian inilah kita menentukan tipe data dari variabel tersebut. Tipe data dibutuhkan agar kompiler dapat mengetahui bagaimana sebuah data akan digunakan. Tipe data dari contoh variabel di atas adalah String. Karena Kotlin mendukung *type inference* maka kita diperbolehkan untuk tidak menuliskan tipe data secara eksplisit:  
  + val company = "Dicoding"
* **Initialization**Dan yang terakhir adalah *initialization* atau nilai awal dari sebuah variabel. Pada contoh di atas yang berperan sebagai *initialization* adalah **“Dicoding”** dan **“Dicoding Academy”**.

Tipe data juga menentukan operasi apa saja yang dapat dilakukan pada sebuah variabel dan bagaimana nilai dari sebuah variabel disimpan. Contoh, ketika kita menggunakan operator + terhadap dua variabel yang bertipe String seperti berikut:

* fun main() {
* val firstWord = "Dicoding "
* val lastWord = "Academy"
* print(firstWord + lastWord)
* }
* /\*
* output: Dicoding Academy
* \*/

Maka kedua nilai dari variabel firstWord dan lastWord akan digabungkan menjadi satu nilai. Berbeda ketika kita menggunakan operator + pada variabel yang bertipe Int seperti berikut:

* fun main() {
* val valueA: Int = 10
* val valueB = 20
* print(valueA + valueB)
* }
* /\*
* output: 30
* \*/

Kompiler akan menjalankan operasi aritmatika, seperti pada contoh di atas di mana nilai dari variabel valueA dan valueB akan dijumlahkan lalu menghasilkan nilai baru.

### **Type Data Char**

Ketika kita mengembangkan sebuah program kita pasti membutuhkan variabel dengan tipe data yang mampu menyimpan nilai berbentuk teks. Terdapat dua (2) tipe data yang bisa kita gunakan, yaitu **Char** dan **String**.

### **Char**

Characters direpresentasikan menggunakan tipe Char. Untuk mendefinisikan sebuah variabel dengan tipe data Char kita bisa menggunakan tanda kutip tunggal **(' ')** seperti berikut:

* val character = 'A'

Tipe data Char hanya dapat kita gunakan untuk menyimpan karakter tunggal. Sebaliknya jika kita memasukkan lebih dari 1 (satu) karakter, akan terjadi eror:

* val character: Char = 'ABC' // Incorrect character literal

Yang menarik, kita bisa melakukan operasi *increment* (++) dan *decrement* (--) pada sebuah variabel dengan tipe data Char seperti berikut:

* fun main() {
* var vocal = 'A'
* println("Vocal " + vocal++)
* println("Vocal " + vocal++)
* println("Vocal " + vocal++)
* println("Vocal " + vocal--)
* println("Vocal " + vocal--)
* println("Vocal " + vocal--)
* println("Vocal " + vocal--)
* }
* /\*
* output:
* Vocal A
* Vocal B
* Vocal C
* Vocal D
* Vocal C
* Vocal B
* Vocal A
* \*/

Operasi *increment* dan *decrement* sendiri merupakan operasi yang bisa kita gunakan pada tipe data **Number**. Lalu kenapa kita bisa menggunakannya pada tipe Char? Karena pada dasarnya setiap Characters merupakan representasi dari **Unicode**. Contoh Unicode **A** adalah **0041.** Ketika kita melakukan *increment* maka hasilnya adalah **0042** yang mana merupakan Unicode dari **B**.

## **3. Tipe Data String**

String merupakan tipe data yang mirip dengan Char. Ia dapat digunakan untuk menyimpan nilai berupa teks. Perbedaannya, String bisa menampung beberapa karakter di dalamnya.

String direpresentasikan menggunakan tipe String. Nilai yang berada di dalam sebuah variabel dengan tipe data String merupakan kumpulan dari beberapa karakter. Kita bisa mendefinisikan variabel tersebut dengan tanda petik ganda (**" "**) seperti berikut:

* val textString = "Kotlin"

Pada dasarnya sekumpulan karakter dalam String tersebut berbentuk Array, sehingga kita bisa mendapatkan karakter tunggal dengan mudah. Caranya, manfaatkan **indexing** seperti berikut:

* fun main() {
* val text = "Kotlin"
* val firstChar = text[0]
* print("First character of $text is $firstChar")
* }
* /\*
* output : First character of Kotlin is K
* \*/

| **Apa itu Indexing?**  *Indexing* merupakan sebuah cara yang memudahkan kita untuk mengakses **elemen** yang berada di dalam sebuah Collection dengan memanfaatkan *index* atau posisi dari elemen tersebut. Posisi dari sebuah elemen pada umumnya dimulai dari angka **0**. Untuk materi tentang Collection akan sama-sama kita pelajari pada sub-modul berikutnya. Semangat! |
| --- |

Nilai **0** yang berada pada *indexing* di atas adalah posisi karakter yang akan diakses. Selain itu, kita juga dapat melakukan iterasi terhadap objek String dengan menggunakan **for-loop** seperti berikut:

* fun main() {
* val text = "Kotlin"
* for (char in text){
* print("$char ")
* }
* }
* /\*
* output : K o t l i n
* \*/

### **Escaped String**

Kotlin memiliki dua jenis tipe **Literal String**, yang pertama adalah **Escaped String** yang memungkinkan kita untuk mengurangi ambiguitas nilai yang berada di dalam sebuah String. Misalnya ketika kita mendefinisikan sebuah String berikut:

* val statement = "Kotlin is Awesome!"

Kemudian kita ingin menambahkan tanda petik ganda di dalam sebuah String seperti berikut:

* val statement = "Kotlin is "Awesome!""

Maka akan terjadi ambiguitas nilai pada variabel statement karena kompiler tidak dapat mengetahui akhir dari baris nilai untuk variabel statement. Untuk mengatasinya, kita bisa melakukan *escaped* dengan menambahkan karakter *backslash* (**\**) sebelum tanda petik ganda seperti berikut:

* val statement = "Kotlin is \"Awesome!\""

Selain **\”** di atas, terdapat beberapa karakter lain yang dapat digunakan untuk melakukan *escaped* di dalam sebuah String, antara lain:

* **\t**: menambah tab ke dalam teks.
* **\n**: membuat baris baru di dalam teks.
* **\’**: menambah karakter *single quote* kedalam teks.
* **\”**: menambah karakter *double quote* kedalam teks.
* **\\**: menambah karakter *backslash* kedalam teks.

Selain itu, kita juga bisa menambahkan sebuah **Unicode** ke dalam sebuah String seperti berikut:

* fun main() {
* val name = "Unicode test: \u00A9"
* print(name)
* }
* /\*
* output: Unicode test : ©
* \*/

### 

### **Raw String**

Kedua, adalah **Raw String** yang memungkinkan kita menuliskan *multiline* dan *arbitrary text*. Ketika ingin membuat beberapa baris String biasanya kita melakukan *escaped* terhadap String dengan memanfaatkan karakter *escape* **\n** seperti berikut:

* val line = "Line 1\n" +
* "Line 2\n" +
* "Line 3\n" +
* "Line 4\n"

Dengan Raw String, kita dapat membuatnya dengan cara yang lebih mudah yaitu seperti berikut:

* fun main() {
* val line = """
* Line 1
* Line 2
* Line 3
* Line 4
* """.trimIndent()
* print(line)
* }
* /\*
* output:
* Line 1
* Line 2
* Line 3
* Line 4
* \*/

Pada kode di atas, kita mendefinisikan sebuah Raw String menggunakan *triple quote* (**""" """**). Raw String memungkinkan kita untuk membuat beberapa baris String tanpa penggabungan (*concatenation*) dan penggunaan karakter *escaped*.

# **4. Functions**

Function atau fungsi merupakan sebuah prosedur yang memiliki keterkaitan dengan pesan dan objek. Ketika kita memanggil sebuah fungsi maka sebuah *mini-program* akan dijalankan. Fungsi sendiri bisa diartikan sebagai cara sederhana untuk mengatur program buatan kita.

Sebuah fungsi dapat kita gunakan untuk mengembalikan nilai. Pemanggilan sebuah fungsi sendiri, bisa diberi argumen atau tidak. Pada sub-modul ini kita akan belajar bagaimana membuat sebuah fungsi pada Kotlin dan mencoba beberapa poin di atas.

* fun functionName(param1: Type1, param2: Type2, ...): ReturnType {
* return result
* }

Pendeklarasian fungsi pada Kotlin diawali dengan kata kunci fun kemudian dilanjutkan dengan nama fungsi yang dikehendaki. Selanjutnya adalah parameter yang berada pada fungsi yang dideklarasikan. Awali dengan nama parameter dan ikuti dengan tipe parameter itu sendiri yang dipisahkan oleh karakter colon (:). Setiap parameter yang berada pada sebuah fungsi dipisahkan oleh karakter koma dan berada di dalam tanda kurung.

* fun setUser(name: String, age: Int)

Setelah menentukan nama dan parameter, selanjutnya adalah menentukan tipe kembalian dari fungsi yang dibuat. Perlu diketahui fungsi pada Kotlin selalu mengembalikan nilai. Tipe kembalian adalah nilai yang akan dikembalikan ketika fungsi tersebut dipanggil.

* fun setUser(name: String, age: Int): **String**

Fungsi di atas akan mengembalikan nilai berupa String. Setelah menentukan tipe nilai kembalian, barulah kita menentukan *function body* di mana di dalamnya terdapat *expression* atau *statement* untuk dijalankan. *Function body* berada di dalam *curly braces* ({}) setelah tipe nilai kembalian.

* fun setUser(name: String, age: Int): String **{**
* **return "Your name is $name, and you $age years old"**
* **}**

Nilai yang akan dikembalikan diikuti oleh kata kunci return. Jika di dalam suatu fungsi hanya memiliki satu *expression* untuk menentukan nilai kembalian, maka fungsi tersebut bisa diubah menjadi *expression body*. Kita hanya perlu menambahkan tanda = dan menuliskannya seperti berikut:

* fun setUser(name: String, age: Int): String **=** "Your name is $name, and you $age years old"

Dengan *expression body*, kompiler dapat menentukan tipe kembalian dari fungsi yang dibuat. Sehingga kita tidak perlu menentukan tipe nilai kembalian secara eksplisit:

* fun setUser(name: String, age: Int) = "Your name is $name, and you $age years old"

Jika kita tidak ingin fungsi yang dibuat mengembalikan nilai, kita bisa menggunakan Unit sebagai tipe nilai kembaliannya. Contohnya seperti berikut:

* fun printUser(name: String): **Unit** {
* print("Your name is $name")
* }

Ketika menggunakan tipe kembalian Unit, Kotlin memungkinkan kita untuk menghilangkannya. Kenapa demikian? Kompiler akan mendeteksinya sebagai tipe kembalian yang *redundant*:

* fun printUser(name: String) {
* print("Your name is $name")
* }

Pemanggilan fungsi, bisa dilakukan dengan pendekatan tradisional seperti berikut:

* fun main() {
* val user = setUser("Alfian", 19)
* println(user)
* printUser("Alfian")
* }
* fun setUser(name: String, age: Int) = "Your name is $name, and you $age years old"
* fun printUser(name: String) {
* println("Your name is $name")
* }
* /\*
* output :
* Your name is Alfian, and you 19 years old
* Your name is Alfian
* \*/

# **5. If Expressions**

Saat mengembangkan sebuah program, kita pasti bertemu dengan alur program yang perlu sebuah kondisi untuk menjalankan sebuah *statement* atau *expression*. Contoh ketika kita ingin menginisialisasi nilai dari sebuah variabel berdasarkan suatu kondisi. Untuk menyelesaikannya, gunakan **If Expression**.

If expression direpresentasikan dengan kata kunci if. If akan kita perlukan untuk menyelesaikan kasus di atas, dimana if akan digunakan untuk menguji suatu kondisi untuk menjalankan sebuah proses. If akan mengeksekusi sebuah *statement* atau *expression* jika hasil evaluasi dari *expressions* yang diberikan pada blok if bernilai **true**. Sebaliknya, jika bernilai **false** maka proses yang ditentukan akan dilewatkan.

* val openHours = 7
* val now = 20
* if (now > openHours){
* println("office already open")
* }

Kode di atas adalah contoh sederhana penggunaan if dengan memanfaatkan operator ***greater than*** untuk membandingkan nilai. Jika if digunakan untuk mengembalikan nilai atau menetapkan nilai dari sebuah variabel maka if wajib memiliki *branch* else. Contohnya seperti berikut:

* val openHours = 7
* val now = 20
* val office: String
* if (now > openHours) {
* office = "Office already open"
* } else {
* office = "Office is closed"
* }
* print(office)

Else akan dijalankan jika hasil evaluasi pada *expression* yang diberikan menghasilkan nilai **false**. If merupakan sebuah *expressions* yang dapat mengembalikan nilai, sehingga kita dapat menyimpan hasilnya ke dalam sebuah variabel.

* val openHours = 7
* val now = 20
* val office: String
* office = if (now > openHours) {
* "Office already open"
* } else {
* "Office is closed"
* }
* print(office)

Pada kode di atas, kita hanya menggunakan If untuk menguji 2 (dua) kondisi. Lalu bagaimana jika kita memiliki beberapa kondisi? Kita bisa menggabungkan else dan if seperti berikut:

* val openHours = 7
* val now = 7
* val office: String
* office = if (now > 7) {
* "Office already open"
* } else if (now == openHours){
* "Wait a minute, office will be open"
* } else {
* "Office is closed"
* }
* print(office)

Blok else if akan dijalankan jika hasil evaluasi pada *branch* sebelumnya bernilai **false**. Jika hasil evaluasi pada *branch* else if juga bernilai nilai **false**, maka lanjut ke evaluasi *branch* selanjutnya.

Perlu diketahui bahwa Kotlin tidak mendukung *ternary operator* (**condition ? then : else**), karena peran dari operator tersebut sudah digantikan dengan if expressions.

**6. Boolean**

Pada sub-modul sebelumnya kita telah belajar tentang **If expressions** yang menggunakan **Boolean expressions**. Kini saatnya kita belajar apa itu **Boolean?** Boolean adalah sebuah tipe data yang hanya memiliki dua nilai, yaitu **true** dan **false**. Terdapat 3 (tiga) operator yang dapat digunakan pada Boolean.

## **Conjunction atau AND (&&)**

Operator AND (&&) akan mengembalikan nilai **true** jika semua hasil evaluasi *expression* yang diberikan bernilai **true**.

* fun main() {
* val officeOpen = 7
* val officeClosed = 16
* val now = 20
* val isOpen = if (now >= officeOpen && now <= officeClosed){
* true
* } else {
* false
* }
* print("Office is open : $isOpen")
* /\*
* Output : Office is open : false
* \*/
* }

Fungsi di atas menguji apakah jam sekarang berada di antara jam waktu buka kantor dan jam tutup kantor. If expressions di atas bisa Anda sederhanakan jadi seperti berikut:

* fun main() {
* val officeOpen = 7
* val officeClosed = 16
* val now = 20
* val isOpen = now >= officeOpen && now <= officeClosed
* print("Office is open : $isOpen")
* /\*
* Output : Office is open : false
* \*/
* }

## **Disjunction atau OR (||)**

Berbeda dengan operator AND (&&), operator OR (||) akan mengembalikan nilai **true** jika hasil evaluasi dari salah satu *expressions* yang diberikan bernilai **true**.

* fun main() {
* val officeOpen = 7
* val officeClosed = 16
* val now = 20
* val isClose = now < officeOpen || now > officeClosed
* print("Office is closed : $isClose")
* /\*
* Output : Office is closed : true
* \*/
* }

Variabel isClose di atas bernilai **true.** Alasannya, hasil evaluasi salah satu *expression* yang diberikan, bernilai **true**, yaitu *expression* disebelah kanan.

## **Negation atau NOT (!)**

Berbeda dengan operator AND (&&) dan operator OR(||), operator NOT(!) digunakan untuk melakukan negasi pada hasil evaluasi *expression* yang diberikan. Contoh, Jika hasil *expressions* setelah dievaluasi bernilai true, maka operator NOT akan mengembalikan nilai **false**.

* fun main() {
* val officeOpen = 7
* val now = 10
* val isOpen = now > officeOpen
* if (!isOpen) {
* print("Office is closed")
* } else {
* print("Office is open")
* }
* /\*
* Output : Office is open
* \*/
* }

Hasil evaluasi *expression* di atas adalah **true.** Tapi ketika menggunakan operator NOT maka akan dinegasikan menjadi nilai **false**. Sehingga statement pada *branch* else-lah yang akan dijalankan.

# **7. Numbers**

Pada sub-modul tipe data kita sudah mempelajari tentang beberapa tipe seperti **Character** dan **String** . Sekarang kita akan mempelajari beberapa tipe data yang termasuk ke dalam tipe **Number**. Number adalah sebuah tipe data yang khusus digunakan untuk menyimpan nilai dalam bentuk numerik.

Di Kotlin, tipe data Number disimpan dengan cara yang berbeda. Beberapa tipe bawaan yang merepresentasikan Numbers adalah **Double**, **Long**, **Int**, **Short** dan **Byte**. Setiap tipe data Number memiliki ukuran (satuan Bit) berbeda-beda, tergantung besaran nilai yang dapat simpan.

* **Int (32 Bit)**Int adalah tipe data yang umumnya digunakan untuk menyimpan nilai numerik. Int dapat menyimpan data dari *range* -2^31 sampai +2^31-1. Dengan ukuran 32 Bit kita bisa menggunakannya untuk menyimpan nilai yang besar. Catatannya, tetap lihatlah batasan nilai maksimal yang dapat dimasukkan.
  + val intNumber = 100
* **Long (64 Bit)**Long adalah tipe data yang digunakan untuk menyimpan nilai numerik yang lebih besar yaitu dari *range* -2^63 sampai +2^63-1. Long bisa didefinisikan secara eksplisit:
  + val longNumber: Long = 100
* Atau dengan menambahkan *suffix* **L** seperti berikut:
  + val longNumber = 100L
* **Short (16 Bit)**Short merupakan sebuah bilangan bulat yang hanya dapat menyimpan nilai yang kecil karena hanya berukuran 16 Bit.
  + val shortNumber: Short = 10
* **Byte (8 Bit)**Dengan ukuran yang kecil, Byte hanya mampu menyimpan nilai yang kecil sama halnya seperti Short. Byte biasa digunakan untuk keperluan proses membaca dan menulis data dari sebuah *stream file* atau jaringan.
  + val byteNumber = 0b11010010
* **Double (64 Bit)**Sama halnya dengan Long yang memiliki ukuran yang besar, Double mampu menyimpan nilai numerik yang besar pula. Pada umumnya Double digunakan untuk menyimpan nilai numerik pecahan sampai dengan maksimal 15-16 angka di belakang koma.
  + val doubleNumber: Double = 1.3
* **Float (32 Bit)**Sama seperti Double, namun memiliki ukuran yang lebih kecil, yakni hanya sampai 6-7 angka di belakang koma.
  + val floatNumber: Float = 0.123456789f //yang terbaca hanya 0.1234567

Untuk mengetahui nilai maksimal yang dapat disimpan oleh suatu tipe Number, kita bisa menggunakan properti MAX\_VALUE. Sementara untuk mengetahui nilai minimal yang dapat disimpan, gunakan properti MIN\_VALUE.

* fun main() {
* val maxInt = Int.MAX\_VALUE
* val minInt = Int.MIN\_VALUE
* println(maxInt)
* println(minInt)
* /\*
* output :
* 2147483647
* -2147483648
* \*/
* }

Jika kita memasukan nilai melebihi nilai maksimal yang dapat disimpan, maka akan terjadi *overflow.* Nilai yang akan dikembalikan adalah nilai minimal yang dapat disimpan.

* fun main() {
* val maxInt = Int.MAX\_VALUE
* val overRangeInt = Int.MAX\_VALUE + 1 // This operation has led to an overflow
* println("Max Int: $maxInt")
* println("Over range Int: $overRangeInt")
* }
* /\*
* Output :
* Max Int: 2147483647
* Over range Int: -2147483648
* \*/

Terdapat beberapa operator matematika pada tipe data Number seperti penjumlahan (**+**), pengurangan (**-**), perkalian (**\***) , pembagian (**/**) dan modulus (**%**, atau sisa hasil bagi).

* // main function
* fun main() {
* val numberOne = 1
* val numberTwo = 2
* print(numberOne + numberTwo)
* /\*
* output : 3
* \*/
* }

Perlu diketahui, hasil operasi pembagian pada tipe data Int akan dibulatkan kebawah. Contohnya seperti berikut:

* // main function
* fun main() {
* val numberOne: Int = 27
* val numberTwo: Int = 10
* print(numberOne / numberTwo)
* /\*
* output : 2
* \*/
* }

Sama seperti perhitungan matematika di mana operasi perkalian dan pembagian didahulukan, demikian halnya perhitungan pada Kotlin.

* fun main() {
* print(5 + 4 \* 4)
* /\*
* output: 21
* \*/
* }

Operasi **4 \* 4** akan dilakukan terlebih dahulu, kemudian diikuti **5 + 16**. Jika ingin operasi **5 + 4** dilakukan terlebih dahulu, gunakan tanda kurung:

* fun main() {
* print((5 + 4) \* 4)
* /\*
* output: 36
* \*/
* }

Di Kotlin kita tidak bisa melakukan konversi secara langsung. Contoh, ketika ingin melakukan konversi dari tipe data Byte ke tipe data Int.

* fun main() {
* val byteNumber: Byte = 1
* val intNumber: Int = byteNumber // compile error
* }

Kode akan gagal dikompilasi dengan log eror berikut:

***Error:(4, 18) Kotlin: Type mismatch: inferred type is Byte but Int was expected***

Untuk mengatasinya, lakukan konversi dengan bantuan beberapa fungsi seperti toInt() berikut:

* fun main() {
* val byteNumber: Byte = 10
* val intNumber: Int = byteNumber.toInt() // ready to go
* }

Kode di atas menggunakan fungsi toInt() untuk melakukan konversi secara eksplisit dari tipe data Byte ke tipe data Int. Adapun beberapa fungsi konversi yang dapat kita gunakan antara lain:

* **toByte(): Byte**
* **toShort(): Short**
* **toInt(): Int**
* **toLong(): Long**
* **toFloat(): Float**
* **toDouble(): Double**
* **toChar(): Char**

Contoh lain penggunaan konversi adalah sebagai berikut:

* fun main() {
* val stringNumber = "23"
* val intNumber = 3
* print(intNumber + stringNumber.toInt())
* /\*
* output: 26
* \*/
* }

Dengan fungsi konversi di atas, nilai **23** yang semula bertipe String di konversi ke tipe Int yang kemudian dimasukan ke dalam operasi matematika.

Dengan Kotlin kita juga bisa menuliskan nilai numerik yang “*readable*” dengan menggunakan tanda *underscores* seperti berikut:

* fun main() {
* val readableNumber = 1\_000\_000
* print(readableNumber)
* /\*
* output : 1000000
* \*/
* }

## **8. Arrays**

Selanjutnya adalah Array, yakni tipe data yang memungkinkan kita untuk menyimpan beberapa objek di dalam sebuah variabel. Array di Kotlin direpresentasikan oleh kelas Array yang memiliki fungsi get dan set serta properti size. Untuk membuat sebuah Array kita bisa memanfaatkan sebuah library function arrayOf() seperti berikut:

* val array = arrayOf(1, 3, 5, 7)

Kita juga dapat memasukkan nilai dengan berbagai jenis tipe data ke dalam arrayOf() misalnya:

* val mixArray = arrayOf(1, 3, 5, 7 , "Dicoding" , true)

Kotlin juga memungkinkan kita untuk membuat Array dengan tipe data primitif dengan memanfaatkan beberapa fungsi spesifik berikut:

* **intArrayOf() : IntArray**
* **booleanArrayOf() : BooleanArray**
* **charArrayOf() : CharArray**
* **longArrayOf() : LongArray**
* **shortArrayOf() : ShortArray**
* **byteArrayOf() : ByteArray**

Jika kita ingin membuat Array yang hanya bisa dimasukkan nilai dengan tipe data **Int,** gunakan intArrayOf(), misalnya:

* val intArray = intArrayOf(1, 3, 5, 7)

Kita juga bisa mendapatkan nilai tunggal dari sekumpulan nilai yang berada di dalam sebuah Array dengan memanfaatkan *indexing* seperti berikut:

* fun main() {
* val intArray = intArrayOf(1, 3, 5, 7)
* print(intArray[2])
* }
* /\*
* Output: 5
* \*/

Nilai **2** pada kode di atas merupakan indeks atau posisi dari nilai tunggal yang ingin kita dapatkan. Perlu diketahui bahwa sebuah indeks selalu dimulai dari **0**. Selain mendapatkan nilai tunggal, dengan *indexing* kita juga bisa mengubah nilai tunggal tersebut. Sebagai contoh:

* fun main() {
* val intArray = intArrayOf(1, 3, 5, 7) // [1, 3, 5, 7]
* intArray[2] = 11 // [1, 3, 11, 7]
* print(intArray[2])
* }
* /\*
* Output: 11
* \*/

Selain menggunakan *library function* arrayOf() dalam pembuatan objek Array, kita juga bisa menggunakan Array(). Constructor pada Array() membutuhkan 2 argumen yaitu size dan fungsi lambda. Contoh untuk membuat Array menggunakan Array() adalah seperti berikut:

* val intArray = Array(4, { i -> i \* i }) // [0, 1, 4, 9]

Pada kode di atas kita menentukan angka **4** sebagai *size* Array. Fungsi lambda di atas ada dua. Pertama, untuk mengambil indeks Array yang akan digunakan sebagai argumen. Kedua, menentukan elemen Array yang akan dimasukkan ke dalam indeks tersebut.

| **Apa itu library Function?**  *Library Function* merupakan sekumpulan fungsi bawaan [**Kotlin Standart Library**](https://kotlinlang.org/api/latest/jvm/stdlib/) yang dapat digunakan untuk menerapkan suatu komponen dengan cara yang sederhana dengan memanfaatkan Lambda Expression. |
| --- |

## **9. Nullable Types**

Ketika mengembangkan sebuah program, ada satu hal yang tak boleh kita abaikan. Ia adalah **NullPointerException (NPE)**, sebuah kesalahan yang terjadi saat ingin mengakses atau mengelola nilai dari sebuah variabel yang belum diinisialisasi atau variabel yang bernilai **null**. Karena sangat umum terjadi dan bisa berakibat fatal, **NPE** terkenal dengan istilah ***“The Billion Dollar Mistake”***.

Dalam penanganannya, kita harus berhati-hati karena NPE menyebabkan aplikasi yang kita kembangkan, rusak saat dijalankan.

Pada Kotlin kita dimudahkan untuk mengelola variabel ***nullable*** sehingga dapat meminimalisir terjadinya **NullPointerException**. Kotlin hadir dengan penanganan *nullability* yang mudah. Kotlin mampu membedakan objek yang boleh bernilai **null** dan objek yang tidak boleh bernilai null pada saat objek tersebut dibuat.

* val text: String = null // compile time error

Kotlin akan memaksa kita untuk menentukan nilai awal dari sebuah objek ketika dibuat dan tidak boleh bernilai **null**. Jika ingin sebuah objek bisa bernilai **null**, kita bisa menambahkan tanda ? setelah menentukan tipe dari objek tersebut:

* val text: String? = null // ready to go

Namun kita tidak bisa langsung mengakses atau mengelola nilai dari objek yang sudah kita tandai sebagai *nullable*. Sebagai contoh:

* val text: String? = null
* val textLength = text.length // compile time error

Ketika kita menuliskan kode di atas, maka akan gagal dikompilasi dengan log eror berikut:

***Error:(4, 26) Kotlin: Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?***

Lalu bagaimana cara kita mengakses atau mengelola nilai dari objek yang ditandai sebagai *nullable*? Cara mudahnya, periksa objek tersebut apakah bernilai **null** atau tidak:

* val text: String? = null
* //val textLength = text.length // compile time error
* if (text != null){
* val textLength = text.length // ready to go
* }

Dengan cara tradisional menggunakan **if/else** seperti di atas, kompiler akan mengizinkan kita untuk mengelola nilai dari sebuah variabel yang kita tandai sebagai *nullable*.

Pada sub-modul selanjutnya kita akan mempelajari penanganan objek yang ditandai sebagai *nullable* dengan cara yang lebih mudah. Kita akan menggunakan **Safe Calls** dan **Elvis Operator** di Kotlin.

**10. Safe Calls** dan **Elvis Operator**

## **Safe calls operator (?.)**

Yang pertama akan kita pelajari adalah **Safe Calls**. Seperti namanya, safe call akan menjamin kode yang kita tulis aman dari **NullPointerException**. Dalam menggunakan safe call, kita akan mengganti tanda titik (.) dengan tanda (?.) saat mengakses atau mengelola nilai dari objek *nullable.* Seperti ini:

* val text: String? = null
* text?.length

Dengan safe call, kompiler akan melewatkan proses jika objek tersebut bernilai **null**.

## **Elvis Operator (?:)**

**Elvis operator** memungkinkan kita untuk menetapkan *default value* atau nilai dasar jika objek bernilai **null**.

* val text: String? = null
* val textLength = text?.length ?: 7

Kode di atas sebenarnya sama seperti ketika kita menggunakan **if/else** berikut:

* val textLength = if (text != null) text.length else 7

Elvis akan mengembalikan nilai text.length jika text tidak bernilai **null.** Sebaliknya, jika text bernilai **null** maka *default value* yang akan dikembalikan.

Sebelum lanjut ke sub-modul selanjutnya terdapat satu hal yang perlu diperhatikan dalam penanganan objek *nullable*. Perhatikan penggunaan operator **non-null assertion** (!!), misalnya seperti berikut:

* val text: String? = null
* val textLength = text!!.length // ready to go ???

Dengan menggunakan **non-null assertion** kompiler akan mengizinkan kita untuk mengakses atau mengelola nilai dari sebuah objek *nullable*. Namun penggunaan operator tersebut sangat tidak disarankan karena akan memaksa sebuah objek menjadi **non-null.** Sehingga ketika objek tersebut bernilai **null**, Anda tetap akan berjumpa dengan **NullPointerException**.

# **11. String Template**

Di beberapa sub-modul sebelumnya Anda sudah melihat bagaimana sebuah String ditulis seperti berikut:

* "First character of $text is $firstChar"

Lantas dinamakan apakah mekanisme penulisan String seperti di atas? Kotlin mendukung sebuah fitur bernama **String Template**. Sebuah fitur yang memungkinkan kita untuk menyisipkan sebuah variabel ke dalam sebuah String tanpa *concatenation* (penggabungan objek String menggunakan **+**) seperti berikut:

* fun main() {
* val name = "Kotlin"
* print("My name is " + name)
* }
* /\*
* output : My name is Kotlin
* \*/

Untuk menggunakan string template, kita hanya perlu menambahkan karakter **$** sebelum nama variabel yang akan disisipkan seperti berikut:

* fun main() {
* val name = "Kotlin"
* print("My name is $name")
* }
* /\*
* output : My name is Kotlin
* \*/

Variabel yang dapat disisipkan tidak sebatas String. Kita juga bisa menyisipkan objek lain misal **Int** atau **Double** seperti berikut:

* fun main() {
* val name = "Kotlin"
* val old = 3
* print("My name is $name, im $old years old")
* }
* /\*
* output : My name is Kotlin, im 3 years old
* \*/

Tidak hanya sampai disitu, Anda juga bisa menyisipkan sebuah *expression* ke dalam sebuah string template. Caranya, sisipkan *expression* ke dalam *curly braces* yang diikuti karakter **$**.

* fun main() {
* val hour = 7
* print("Office ${if (hour > 7) "already close" else "is open"}")
* }
* /\*
* output : Office is open
* \*/

Dengan string template, kita lebih mudah membuat objek String yang dinamis.